Take me for a spin

Implementing Locks in C

Credits to Charlotta Spik, Hannes Rabo, Fredrik Carlsson
2019-11-27
KTH Kista, Stockholm

Total Store Order

- Write operations are ordered
- Read operations can move around
  - A read can be done before a write is completed
- So even if a write is performed before a read, the read can be done first
  - We will then read an old value

Peterson’s Algorithm

- Has an array of “flags” with size 2, one flag for each process
- Has a shared global variable “turn” to represent whose turn it is
- When a process wants to execute it’s critical section, it sets its flag to true and turn as the index of the other process
    - The flag indicates that the process wants to enter the critical section, but setting turn to the other process’ index will allow the other process to run first
- While the the other process’ flag is true and it is the other process’ turn, spin
- After a process is done in the critical section, it sets its flag to false
- The other process can now enter the critical section

Peterson’s Algorithm - Problem!

- The algorithm is dependent on that the process first sets its own value, then reads the value of the other process
- Because of total store order, this can be violated!
- Before the write, flag = 1, is complete, the other process can read the value as 0 and assume the other thread is not interested in the critical section
- Both threads will then enter the critical section and overwrite each other’s value

Peterson’s Algorithm - Problem!

- The algorithm is dependent on that the process first sets its own value, then reads the value of the other process
- Because of total store order, this can be violated!
- Before the write, flag = 1, is complete, the other process can read the value as 0 and assume the other thread is not interested in the critical section
- Both threads will then enter the critical section and overwrite each other’s value
- We solve this with atomic operations

The solution: test-and-set

- Atomic operation
- Returns the old value while simultaneously setting the new value
- If no one holds the lock:
  - Simultaneously set value to 1 while returning 0 => success
  - The value is now 1, which means someone has the lock
  - Set the lock to 0 again when done in critical section
- If another thread has the lock:
  - Simultaneously set value to 1 while returning 1 => failure
  - 1 means that another thread has the lock
  - try until test-and-set returns 0

Compare and Swap

- Similar to “test-and-set”
- Check if the lock is 0, and if so, sets it to 1 and acquire the lock
- If not 0, spin until it is

__sync_val_compare_and_swap

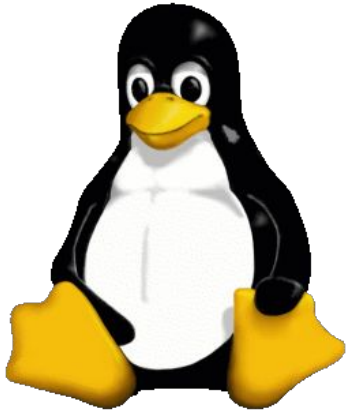
Spinning Wastes Time

- The thread spends all its time slot just waiting for a value to change
- Another solution is to make the thread yield while waiting, and wake up the thread when the value has changed
- For this we use futext()
  - This is a system call that provides a method for waiting until a certain condition becomes true
  - For example when a lock has been released
  - FUTEX_WAIT to put thread to sleep while waiting
  - FUTEX_WAKE to wake up a thread when the condition has changed

Spinning or Sleeping?

- Sometimes, doing a context switch between threads actually takes longer than waiting for the value to change
  - Maybe the lock will soon be released
- This will lead to an unnecessary switch to a new thread that does not get to execute for long before switching back
- In this case, spinlocks are a better choice

UNIX - SYS_futex



- FUTEX - Fast Userspace Mutex Lock
- The way mutexes are handled natively in UNIX
- Not the way to do it yourself - Use the POSIX standard (pthread_mutex)

POSIX Threading - pthreads

- Managing pthreads
  - pthread_create
  - pthread_join

```
pthread_t myThread;
pthread_create(&myThread , NULL, myFunction, "Simple argument");
pthread_join(myThread, NULL);
```

- Threads only takes one input argument
  - Can pass several values to a thread by using a **struct**

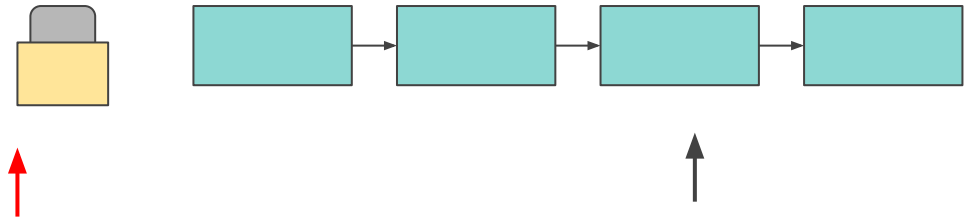
```
typedef struct args {int myInt; char myChar;} args;
args myArgument;
myArgument . myInt = 42;
myArgument . myChar = 'F';

pthread_create(&myThread , NULL, myFunction, &myArgument);
```

POSIX Threading - pthread_mutex

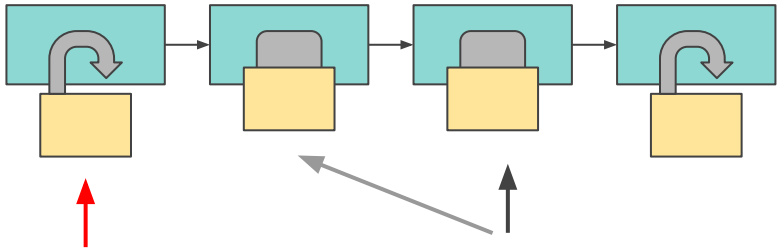
- Managing mutex locks
  - pthread_mutex_lock
  - pthread_mutex_unlock
  - PTHREAD_MUTEX_INITIALIZER

Locking you Lists

One “master lock”



One lock in each node - **Hand-over-Hand-locking** locking



Nifty Swifty Keywords

Compiling and including PThread

- gcc my_program.c -o my_program -lpthread

Waits until specified thread terminates

- pthread_join

Creates a thread for a specified function

- pthread_create

Disable compiler optimizations of a variable

- Volatile

Parse string to int

- Atoi

Virtualbox?

Ensure machine cores are set to >1 core(s) !

Exam Questions

Exam Question 1

You and your colleague Alan Peterson have realized that you need to implement a lock to protect shared data structures. Alan proposes that each thread should have a tag that signals that the thread wants to enter the critical section. If a tread sets its own tag and then checks that no other thread has set its tag the problem is solved. If another thread also has the tag set the tag is reset and tries again sometime later. There is a risk for starvation but in this problem that is acceptable.

What will you tell your colleague?

You and your colleague Alan Peterson have realized that you need to implement a lock to protect shared data structures. Alan proposes that each thread should have a tag that signals that the thread wants to enter the critical section. If a tread sets its own tag and then checks that no other thread has set its tag the problem is solved. If another thread also has the tag set the tag is reset and tries again sometime later. There is a risk for starvation but in this problem that is acceptable.

What will you tell your colleague?

Answer:
Alan, wtf? You know that on a modern processor we can't guarantee sequential consistency, but only guarantee to have **Total Store Order**. Hence there's a chance that we might end up with several threads inside the critical section.

Exam Question 2

Assume we have a ***single linked sorted list*** that we wish to update by adding or removing elements. We want several threads to be able to perform operations in parallel and therefore implement a list where ***each cell is protected by a lock***.

The operations require that we hold two locks when we remove an item but this is easily implemented. We can choose to use ***pthread_mutex_lock()*** to implement the lock operations but we could also use simple spin-locks. ***What is the advantage and disadvantage of using spin-locks and how are the conditions changed if we have more threads and less cores?***

Assume we have a ***single linked sorted list*** that we wish to update by adding or removing elements. We want several threads to be able to perform operations in parallel and therefore implement a list where ***each cell is protected by a lock***.

The operations require that we hold two locks when we remove an item but this is easily implemented. We can choose to use ***pthread_mutex_lock()*** to implement the lock operations but we could also use simple spin-locks. ***What is the advantage and disadvantage of using spin-locks and how are the conditions changed if we have more threads and less cores?***

Answer:

The advantage with spin-locks would be that the locks are held for a short time and a taken lock is probably released within a few clock cycles. If we spin we can ***avoid suspending*** a process.

The down side is of course ***if the process that holds the lock is suspended***. If we increase the number of threads or have fewer cores the risk for this increase which would then be an argument against using spin-locks

Exam Question 3

2.1 a buffer [2 points]

We implement a buffer with one element as shown below (get() defined in similar way). We will have several threads that produce and consume values from the buffer. The buffer is protected by a lock and the threads are synchronized using a conditional variable. The program below does not work (the calls to the pthread procedures are not even legal), why does it not work (even if they where legal)?

```
#define TRUE 1
#define FALSE 0

volatile int buffer = 0;
volatile int empty = TRUE;

pthread_mutex_t lock;
pthread_cond_t signal;

void put(int value) {
  pthread_mutex_lock(&lock);
  while(TRUE) {
    if(empty) {
      buffer = value;
      empty = FALSE;
      pthread_cond_signal(&signal);
      pthread_mutex_unlock(&lock);
      break;
    } else {
      pthread_mutex_unlock(&lock);
      pthread_cond_wait(&signal);
    }
  }
}
```

2.1 a buffer [2 points]

We implement a buffer with one element as shown below (get() defined in similar way). We will have several threads that produce and consume values from the buffer. The buffer is protected by a lock and the threads are synchronized using a conditional variable. The program below does not work (the calls to the pthread procedures are not even legal), why does it not work (even if they where legal)?

```
#define TRUE 1
#define FALSE 0

volatile int buffer = 0;
volatile int empty = TRUE;

pthread_mutex_t lock;
pthread_cond_t signal;

void put(int value) {
  pthread_mutex_lock(&lock);
  while(TRUE) {
    if(empty) {
      buffer = value;
      empty = FALSE;
      pthread_cond_signal(&signal);
      pthread_mutex_unlock(&lock);
      break;
    } else {
      pthread_mutex_unlock(&lock);
      pthread_cond_wait(&signal);
    }

}
```

Answer: The problem is that we release the lock and then suspend on the conditional variable in two operations. If the process is interrupted between these operations, another process might: call get(), take the lock, remove the item from the buffer and signal. If we now suspend on the conditional we will never wake up.

The code also has the problem that if we do wake up the lock is not held. We also have a problem that a producer can wake up another producer instead of a consumer.

Exam Question 4

1.3 `__sync_val_compare_and_swap()` [2 points*]

We can implement a spin lock in GCC as shown below. The lock is implemented using a machine instruction that atomically will read the content of a memory location and, if it is equal to our requirement, replace it with a new value. In the implementation below we represent an open lock with the value 0; if the lock is open we write a 1 in the location and return 0, otherwise we return the value found (that then should be 1).

Assume that we use the lock to synchronize two threads on a machine with only one core; what is then the disadvantage that we will have? How could we mitigate the problem?

```
int try(volatile int *mutex) {
  return __sync_val_compare_and_swap(mutex, 0, 1);
}

void lock(volatile int *mutex) {
  while(try(mutex) != 0) { }
}

void release(volatile int *mutex) {
  *mutex = 0;
}
```

1.3 `__sync_val_compare_and_swap()` [2 points*]

We can implement a spin lock in GCC as shown below. The lock is implemented using a machine instruction that atomically will read the content of a memory location and, if it is equal to our requirement, replace it with a new value. In the implementation below we represent an open lock with the value 0; if the lock is open we write a 1 in the location and return 0, otherwise we return the value found (that then should be 1).

Assume that we use the lock to synchronize two threads on a machine with only one core; what is then the disadvantage that we will have? How could we mitigate the problem?

```
int try(volatile int *mutex) {
  return __sync_val_compare_and_swap(mutex, 0, 1);
}

void lock(volatile int *mutex) {
  while(try(mutex) != 0) { }
}

void release(volatile int *mutex)
  *mutex = 0;
}
```

Answer: If a thread takes the lock and is then suspended by the scheduler, the scheduled thread could in the worst case spend its whole allotted time slot spinning. We could yield the CPU, `pthread\_yield()`, to allow the suspended thread to continue its execution.